1 Data Strukturer

1.1 Linked List

Methods A `LinkedList` has the following methods

Method	Runtime
`add(E elem)`	$O(1)$ for all indices
`contains(E e)`	$O(n)$
`remove(E e)`	$O(n)$
`removeFirst(E e)`	$O(1)$
`removeLast(E e)`	$O(1)$
`toArray()`	$O(n)$
`indexOf(E e)`	$O(n)$
`get()`	$O(n)$
`set(int i, E e)`	$O(n)$

1.2 Array List

An `ArrayList` has the following methods

Method	Runtime
`add(int index, E elem)`	$O(n)$
`add(E elem)`	$O(n)$
`contains(E e)`	$O(n)$
`remove(E e)`	$O(n)$
`toArray()`	$O(n)$
`indexOf(E e)`	$O(n)$
`get`	$O(1)$
`set(int i, E e)`	$O(1)$

1.3 Heap (Priority Queue)

A heap, or Priority Queue, implements the following methods

Method	Runtime
`add(E e)`	$O(\log n)$
`offer(E e)`	$O(\log n)$
`remove(E e)`	$O(\log n)$
`poll()`	$O(\log n)$
`peek()`	$O(1)$
`element()`	$O(1)$
`contains(E e)`	$O(n)$

1.4 Hash-tabell (HashMap)

1.5 Binært Søketre

1.6 Union Find

A Union Find, or Disjoint Set Union data structure is used in Kruskal's algorithm in order to find the minimum spanning tree.

```java
class UnionFind <V> {
   public HashMap<V, DSUNode<V>> components = new HashMap<>();

   public void makeSet(V v) {
      if (!components.containsKey(v)) {
         components.put(v, new DSUNode<V>(v, v));
      }
   }

    public boolean connected(V v, V u) {
        return find(v).equals(find(u));
    }

   public V find(V v) {
      DSUNode<V> curr = components.get(v);
      if (!curr.parent.equals(curr.node)) {
         components.put(v, new DSUNode<V>(find(curr.parent), v));
         return components.get(v).parent;
      }
      return v;
   }

   public void union(V v, V u) {
      V x = find(v);
      V y = find(u);

      if (!x.equals(y)) {
            components.put(y, new DSUNode<V>(x, y));
      }
   }
}

class DSUNode<V> {
   public V parent;
   public V node;

   public DSUNode(V parent, V node) {
      this.parent = parent;
      this.node = node;
   }
}
```

1.7 Graf

2 Algoritmer

2.1 Søke Algoritmer

2.1.1 Binary Search

A binary search algorithm is a searching algorithm that finds the index of the element sent in as argument to the function. A prerequisite for the binary search algorithm is that the list we are performing the search on is **sorted**.

Runtime The runtime of the binary search algorithm is $O(\log(n))$.

Below is an implementation of the binary search algorithm.

```java
public class BinarySearch<E extends Comparable<? super E>> {
    List<E> sorted;

    public BinarySearch(ArrayList<E> sorted) {
        this.sorted = sorted;
    }

    public int search(E target) {
        int lo = 0;
        int hi = sorted.size() - 1;
        int mid = 0;

        while (lo <= hi) {
            mid = hi - (hi - lo) / 2;

            if (sorted.get(mid).compareTo(target) < 0) {
                lo = mid + 1;
            } else if (sorted.get(mid).compareTo(target) > 0) {
                hi = mid - 1;
            } else {
                return mid;
            }
        }
        throw new NoSuchElementException("Element not in list");
    }
}
```

2.1.2 Quick Select

`QuickSelect` is a selection algorithm to find the `kth` smallest (or `kth` largest) element in a list. Whats special about the `QuickSelect` algorithm, is that it is not necessary that the list we are selecting from is sorted. It is related to the `QuickSort` algorithm in that it selects a `pivot`, and partitions the list based on this pivot. In contrast to the `QuickSort` algorithm, `QuickSelect` only recurses into one of the partitions of the list. It chooses the list which contains the `kth` element.

Runtime The `QuickSelect` algorithm has an average runtime of $O(n)$, and a worst case of $O(n^2)$. The worst case occurs when pivots are chosen poorly, such as only decreaseing the pivot by one every time. This is easily mitigated by choosing a pivot in the of the current search scope at random.

Below is an implementation of the algorithm

```java
public class QuickSelect<T extends Comparable<? super T>> {
    public T select(List<T> list, int left, int right, int k) {
        if (left == right) {
            return list.get(left);
        }

        Random rand = new Random();
        int pivot = rand.nextInt(left, right);
        pivot = partition(list, left, right, pivot);

        if (k == pivot) {
            return list.get(k);
        } else if (k < pivot) {
            return select(list, left, pivot - 1, k);
        } else {
            return select(list, pivot + 1, right, k);
        }
    }

    private int partition(List<T> list, int left, int right, int pivot) {
        T pivotVal = list.get(pivot);
        Collections.swap(list, pivot, right);
        int storeIndex = left;

        for (int i = left; i < right; i++) {
            if (list.get(i).compareTo(pivotVal) < 0) {
                Collections.swap(list, storeIndex++, i);
            }
        }
        Collections.swap(list, right, storeIndex);
        return storeIndex;
    }
}
```

2.2 Sortering

2.2.1 Quick Sort

`QuickSort` is a sorting algorithm that operates on the Divide and Conquer principle. It is a sorting algorith that can be done both in place, and not in place. It works by selecting a pivot in the list, and moves all elements that are smaller than the pivot to the left, and all that are greater to the right. Next, these lists are sorted recursively.

Runtime The `QuickSort` algorithm has a best-case runtime of $O(n \log n)$, and a worst case runtime of $O\left(n^2\right)$.

Below is an implementations of the `QuickSort` algorithm.

```java
public class QuickSort <T extends Comparable<? super T>> {

    public List<T> sort(List<T> list) {
        int left = 0;
        int right = list.size() - 1;
        return qsort(list, left, right);
    }

    private List<T> qsort(List<T> list, int left, int right) {
        if (left >= right || left < 0) {
            return list;
        }

        int pivot = partition(list, left, right);

        qsort(list, left, pivot - 1);
        qsort(list, pivot + 1, right);

        return list;
    }

    private int partition(List<T> list, int left, int right) {
        T pivot = list.get(right);

        int tmp = left - 1;
        for (int i = left; i < right; i++) {
            if (list.get(i).compareTo(pivot) <= 0) {
                Collections.swap(list, ++tmp, i);
            }
        }

        Collections.swap(list, ++tmp, right);
        return tmp;
    }
}
```

2.2.2 Merge Sort

Merge Sort is also a sorting algorithm based on the divide-and-conquer principle. It continusly splits the list into smaller and smaller sublists until they contain only one element, they are then considered sorted. We then continusly merge sublists, making sure they are sorted while we merge them. The final list will then be the sorted list. Merge sort can achieve a runtime of $O(n \log(n))$

```java
public class MergeSort<T extends Comparable<? super T>> {
    public List<T> sort(List<T> list) {
        if (list.size() <= 1) {
            return list;
        }

        List<T> left = new ArrayList<>();
        List<T> right = new ArrayList<>();

        for (int i = 0; i < list.size(); i++) {
            if (i < list.size() / 2) {
                left.add(list.get(i));
            } else {
                right.add(list.get(i));
            }
        }

        left = sort(left);
        right = sort(right);

        return merge(left, right);
    }

    private List<T> merge(List<T> left, List<T> right) {
        List<T> list = new ArrayList<>();

        while (!left.isEmpty() && !right.isEmpty()) {
            if (left.get(0).compareTo(right.get(0)) <= 0) {
                list.add(left.remove(0));
            } else {
                list.add(right.remove(0));
            }
        }

        while (!left.isEmpty()) {
            list.add(left.remove(0));
        }

        while (!right.isEmpty()) {
            list.add(right.remove(0));
        }

        return list;
    }
}
```

2.2.3 Insertion Sort

2.2.4 Selection Sort

2.3 Graf Algoritmer

2.3.1 BFS & DFS

BFS and DFS are graph-searching algorithms for unweighted graph, it is a very versatile algorithm, with a runtime of $O(n)$. These algorithms can be used to find anode furthest from the root, detect cycles, generating depth / height maps for a graph, and topological sorting.

```
public class BFS<V> {
    public V furthest(Graph<V> g, V root) {
        HashSet<V> found = new HashSet<>();
        LinkedList<V> toSearch = new LinkedList<>();
        V furthest = root;

        while (!toSearch.isEmpty()) {
            // --- change to removefirst and we'll have DFS
            V next = toSearch.removeLast();
            if (found.contains(next)) {
                continue;
            }

            found.add(next);
            furthest = next;
            g.neighbours(next).forEach(n -> toSearch.add(n));
        }

        return furthest;
    }
}
```

2.3.2 Minimum Spanning Tree

Kruskal's

```
public class Kruskal {
   public <V, E extends Comparable<E>> List<Edge<V>> mst(WeightedGraph<V, E> g) {
      List<Edge<V>> edges = new ArrayList<>();
      List<Edge<V>> mst = new ArrayList<>();
      int size = g.size();


      // --- create a list of the edges and sort them, then add them to the vertices list
        g.edges().forEach(e -> edges.add(e));
      edges.sort(g); // o(m log (m))
      UnionFind<V> UF = new UnionFind<>();

      for (int i = 0; i < edges.size(); i++) { // o(n^2)
         Edge<V> current = edges.get(i);
         UF.makeSet(current.a);
         UF.makeSet(current.b);

            if (!UF.connected(current.a, current.b)) {
               UF.union(current.a, current.b);
```

```
                mst.add(current);
            }

         if (mst.size() == size - 1) {
            return mst;
         }
      }

      return mst;
   }
}
```

2.3.3 Dijkstra's Algorithm

2.3.4 Bellman-Ford